DESCRIÇÃO

Aplicação de ponteiros, Alocação dinâmica de memória, Estruturas de dados heterogêneas usando a linguagem de programação C.

PROPÓSITO

Compreender o conceito de estruturas de dados heterogêneas e aplicá-las usando a linguagem de programação C, declarando e manipulando os dados a partir de structs, structs aninhadas e array de structs.

PREPARAÇÃO

Antes de iniciar o conteúdo deste tema, você precisa de um compilador C, como o GCC, e uma IDE. Recomendamos que você instale o Dev-C++, que é um ambiente de desenvolvimento integrado livre que utiliza os compiladores do projeto GNU para compilar programas para o sistema operacional Microsoft Windows.

Você pode baixar o código-fonte dos exemplos clicando aqui ou na seção “Explore +” desse tema.

OBJETIVOS

MÓDULO 1

Empregar ponteiros com a utilização da linguagem de programação C

MÓDULO 2

Definir estrutura de dados heterogênea

MÓDULO 3

Aplicar structs com a utilização da linguagem de programação C

MÓDULO 4

Empregar as estruturas de dados aninhadas, os vetores de estruturas e a instrução typedef usando a linguagem de programação C

INTRODUÇÃO

As estruturas de dados definem os diversos tipos de mecanismos que podem ser empregados para organizar os dados que serão tratados pelos algoritmos computacionais para a solução de um problema.

Escolher corretamente a estrutura de dados tem impacto direto no desempenho do algoritmo. Se você escolher uma estrutura inadequada, seu algoritmo terá um desempenho insatisfatório, acarretando uma má experiência.

Portanto, é necessário que estudemos os diversos tipos de estrutura de dados para permitir que você possa selecionar a estrutura mais adequada para a solução de um determinado problema, com um desempenho adequado.

Neste tema, iniciaremos pelo emprego dos ponteiros e, a seguir, conceituaremos estruturas heterogêneas, relembrando a definição de estruturas de dados e as diferenças entre as estruturas homogêneas e heterogêneas.

Aplicaremos os conhecimentos utilizando a linguagem C e desenvolvendo aplicações com ponteiros e structs, além de explorar recursos avançados, como as structs aninhadas e os vetores de structs.

MÓDULO 1

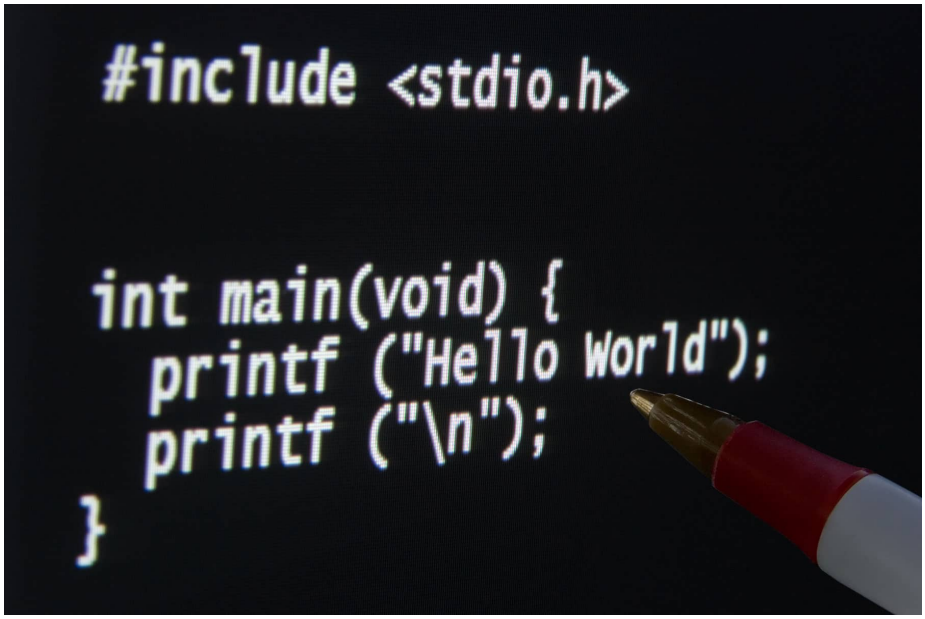
⦿ **Empregar ponteiros com a utilização da linguagem de programação C**

DEFINIÇÃO

Segundo Schildt (1996), ponteiro é uma variável que contém um endereço de memória. Podemos ainda definir o ponteiro como um tipo especial de variável, na qual o valor atribuído é um endereço de memória.

Ou seja, ponteiro ou apontador é uma variável capaz de armazenar um endereço de memória ou o endereço de outra variável.

Para entendermos melhor essa definição, precisamos compreender como a memória de um programa é organizada e o que é um endereço de memória.



Fonte: Shutterstock

MEMÓRIA

Memória é um componente do computador responsável pelo armazenamento de dados e instruções. Ela é composta por palavras, sendo cada palavra identificada unicamente a partir de um endereço, ou seja, um endereço de memória.

Cada palavra tem uma capacidade de armazenamento da informação, isto é, uma quantidade de bytes que a palavra representa.

Sendo assim, aprendemos que uma memória é composta por palavras e que toda palavra possui um endereço único, conforme é visto na Tabela 1.

Endereço	Palavras
0 (00)	Palavra 0
1 (001)	Palavra 1
2 (010)	Palavra 2

3 (011)	Palavra 3
4 (100)	Palavra 4
5 (101)	Palavra 5
6 (110)	Palavra 6

Tabela 1: Representação da memória. Fonte: O Autor.

Atenção! Para visualização completa da tabela utilize a rolagem horizontal

O endereço de memória de um processo em execução é dividido em vários segmentos lógicos. Destacamos alguns dos mais importantes:

TEXT

Contém o código do programa e suas constantes. Este segmento é alocado durante a criação do processo ("exec") e permanece do mesmo tamanho durante toda a vida do processo.

DATA

Este segmento é a memória de trabalho do processo, na qual ficam alocadas as variáveis globais e estáticas. Tem tamanho fixo ao longo da execução do processo.

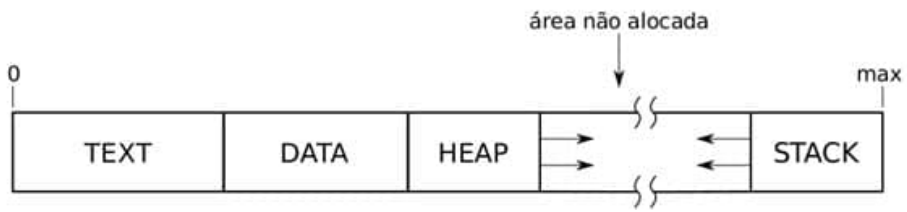
STACK

Contém a pilha de execução, na qual são armazenados os parâmetros, endereços de retorno e variáveis locais de funções. Pode variar de tamanho durante a execução do processo.

HEAP

Contém blocos de memória alocadas dinamicamente, a pedido do processo, durante sua execução. Varia de tamanho durante a vida do processo.

Podemos observar na Figura 1 a distribuição dos segmentos lógicos do endereço de memória.



Fonte:O Autor

Figura 1: Seguimentos lógicos do endereço de memória. Fonte: O Autor.

ALOCAÇÃO DE MEMÓRIA

Agora que já sabemos o que é memória e como ela está organizada internamente, vamos entender as três formas de alocação de memória usando a linguagem C.

Alocação estática (geralmente aloca em DATA).

Alocação automática (geralmente aloca em STACK).

Alocação dinâmica (geralmente aloca em HEAP).

ALOCAÇÃO ESTÁTICA

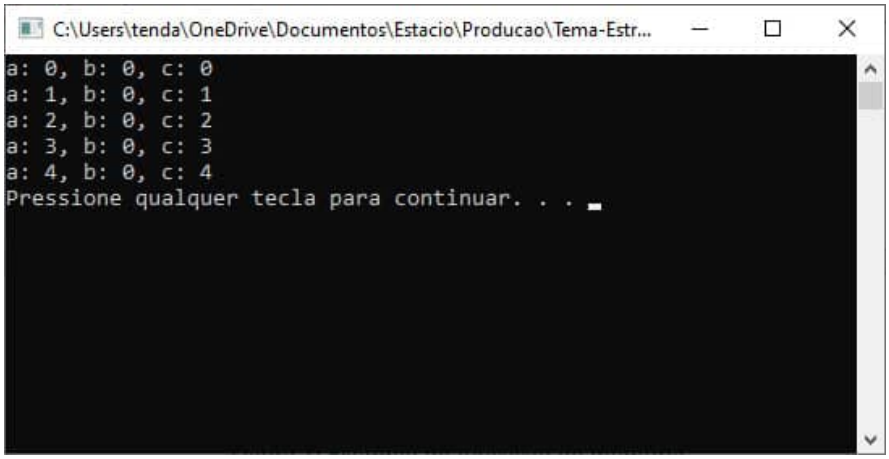
Na **alocação estática**, ocorre a declaração de variáveis globais (alocadas fora de uma função) ou estáticas (alocadas dentro de uma função), usando o modificador “static”. Neste caso, o valor alocado à variável se mantém durante toda a vida do programa, exceto quando explicitamente é modificado.

Analisemos o exemplo 1 para entendermos melhor.

```
1    #include < stdio.h >
2
3    static int a = 0;  // variável global, alocação estática
4
5    void incrementa(void)
6    {
7            int b = 0; // variável local, alocação automática
8            static int c = 0; // variável local, alocação estática
9
10           printf ("a: %d, b: %d, c: %d\n", a, b, c);
11           a++;
12           b++;
13           c++;
14   }
15
16   int main(void)
17   {
18           int i;
19           for (i = 0; i < 5; i++)
20           incrementa();
21           system("pause");
22           return(0);
23   }
```

Exemplo 1: Alocação de memória. Fonte: O Autor.

A execução desse código gera a seguinte saída, conforme mostra a Figura 2.



Fonte:O Autor

Figura 2: Saída da execução do Exemplo 1. Fonte: O Autor.

Analisando este código, podemos observar nas linhas 3 e 8 a declaração da variável “a” e da variável “c”. A variável “a” foi declarada como uma variável global e a variável “c” como uma variável local.

Também é observado que as variáveis foram alocadas de forma estática e inicializadas uma única vez. Desta maneira, seus valores se preservam entre as chamadas consecutivas da função “incrementa”.

Na primeira chamada da função incrementa, a variável “a”, por ser global, já tem o valor 0 e a variável “c” recebe o valor 0 dentro da função. Na linha 10, esse valor é impresso, e logo em seguida as variáveis são incrementadas, passando a possuir o valor 1.

A função incrementa é mais uma vez chamada e podemos observar que as variáveis, por serem estáticas, permanecem com os seus valores, é impresso o valor 1 e são incrementadas mais uma vez, passando a possuir o valor 2.

O programa, deste modo, continua a sua execução até o valor da variável i ser igual a 5, conforme o laço de repetição “FOR”.

ALOCAÇÃO AUTOMÁTICA

Na **alocação automática**, ocorre a declaração de variáveis locais e parâmetros de função. Sendo assim, a alocação dessas variáveis é realizada quando a função é invocada e liberada quando a função termina.

COMENTÁRIO

Voltando a observar o Exemplo 1, a variável “b” é declarada de forma automática ao ser invocada a função incrementa, conforme a linha 7 do código-fonte, e descartada quando a função se encerra.

Neste caso, o valor da variável “b” não se preserva durante a chamada da função incrementa, isto é, o valor alocado à variável não se mantém durante toda a vida do programa.

ALOCAÇÃO DINÂMICA

Na **alocação dinâmica**, é requisitada explicitamente pelo programa uma área de memória para armazenamento de dados. O controle das áreas alocadas dinamicamente pode ser manual ou semiautomático, desta maneira o programa as utiliza e depois as libera quando não forem mais necessárias ou quando o programa encerrar. A liberação das áreas alocadas dinamicamente é realizada pelo programador.

A **alocação dinâmica manual** pode ser realizada de forma **simples** ou **por vetor**.

```
#include < stdlib.h >
void * malloc (size_t size)
```

A função “*malloc*” aloca um determinado número de bytes na memória e retorna um ponteiro para o primeiro byte alocado. Se não for possível alocar, a função retorna “NULL”.

A liberação é realizada pela função “*free*”. Esta função libera o espaço alocado, isto é, libera a área de memória previamente alocada dinamicamente.

```
#include < stdlib.h >
void free (void *ptr)
```

A função “*free*” libera o número de bytes alocados previamente na memória, apontado por “ptr”. O ponteiro “ptr” continua apontando para a área liberada e por isso é aconselhável mudar seu valor para “NULL” após a liberação.

```
ptr = malloc (1024);
...
free (ptr);
ptr = NULL; // não é obrigatório, mas aconselhável
```

O redimensionamento da área alocada é realizado pela função “*realloc*”.

```
#include < stdlib.h >
void * realloc (void *ptr, size_t newsize)
```

A função “*realloc”* redimensiona a área previamente alocada, apontada por “ptr”, para o novo tamanho “*newsize*”. A função retorna o novo endereço da área de memória, que pode ser diferente do anterior, caso tenha sido necessário mudar de lugar.

Em resumo, podemos visualizar cada função com seu objetivo na Tabela 2.

Função	Ação	Sintaxe
Função malloc	A função “malloc” aloca um determinado número de bytes na memória e retorna um ponteiro para o primeiro byte alocado. Se não for possível alocar, a função retorna NULL.	#include < stdlib.h > void *malloc (size_t size)

Função free	A função “free” libera o número de bytes alocados previamente na memória, apontado por “ptr”.	#include < stdlib.h > void free (void *ptr)
Função realloc	A função “*realloc*” redimensiona a área previamente alocada, apontada por “ptr”, para o novo tamanho “*newsize*”.	#include < stdlib.h > void *realloc (void *ptr, size_t newsize)

Tabela 2: Relação de funções utilizadas na alocação dinâmica simples de memória.

Atenção! Para visualização completa da tabela utilize a rolagem horizontal

Na **alocação por vetor**, utilizam-se ponteiros para a alocação.

Sendo assim, basta especificar o tamanho desse vetor no momento da alocação. Após a alocação de uma área com vários elementos, ela pode ser acessada exatamente como se fosse um vetor.

A alocação utiliza a função "*calloc()*" para alocar um bloco de memória de tamanho suficiente para conter um vetor com "count" elementos de tamanho "eltSize" cada um.

```
#include < stdlib.h >
void * calloc (size_t count, size_t eltSize)
```

PONTEIRO

Na linguagem C, cada variável tem um nome, um tipo, um valor e um endereço. Por exemplo, nas variáveis abaixo:

```
int x = 5;
char c = ’D’;
```

Temos que o nome da variável é “x”, o tipo é inteiro, o valor é 5 e essa variável está armazenada na memória no endereço 10. A variável “x” usa dois bytes de memória e quando um objeto usa mais de um byte, seu endereço é onde ele começa, no caso, 10.

Assim como o nome da outra variável é “c”, o tipo char, o valor “D”, está armazenada no endereço de memória 13 e usa um byte de memória.

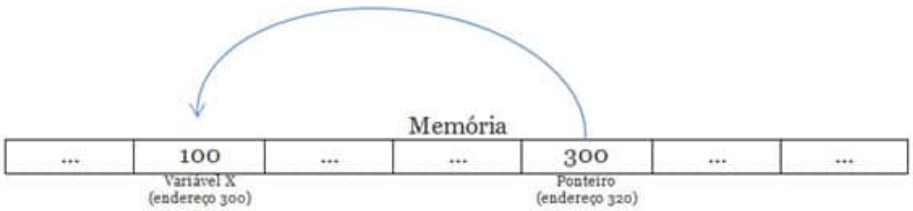
Na Figura 3, podemos observar a alocação dos valores das variáveis x e c na memória.

Fonte:O Autor

Figura 3: Alocação da variável “x” e “c” na memória. Fonte: O Autor.

Já sabemos que todo dado armazenado em memória possui um endereço e que a definição de um ponteiro é uma variável que guarda o endereço de memória, ou seja, a localização do dado.

Sendo assim, um ponteiro armazena o endereço de outra variável, isto é, uma variável que aponta para outra.



Fonte:Shutterstock

Figura 4: Ponteiro armazena endereço da variável “x”. Fonte: O Autor.

A figura 4 apresenta uma memória que armazena a variável “x” e o ponteiro para a variável “x”. Como o conteúdo do ponteiro é um endereço, a seta indica a relação entre o ponteiro e a variável que ele aponta.

No exemplo, o valor da variável “x” é 100 e esse dado está armazenado no endereço de memória 300. Como o ponteiro também ocupa espaço, o endereço de memória da variável “x” está armazenado no endereço 320 da memória, que é a posição de memória alocada para o ponteiro.

Uma das características importantes de um ponteiro é que, utilizando ponteiros, podemos realizar o acesso indireto a outras variáveis, isto é, podemos ler ou alterar o conteúdo de uma variável sem utilizar o nome desta variável.

DECLARAÇÃO

Os ponteiros são declarados pelo símbolo “*” entre o tipo e o nome da variável. A forma geral da declaração é:

Tipo_da_variável * Nome_da_Variável;

Exemplos:

int *p;

*float *q;*

char *r;

As variáveis p, q e r são apontadores (ponteiro) para um inteiro, float e caractere, respectivamente.

OPERADORES

Vejamos dois tipos de operadores:

Operador unário “&” ou ponteiro constante – Tem a função de obter o endereço de memória de uma variável.

Operador unário “*” de indireção – É usado para fazer a deferência.

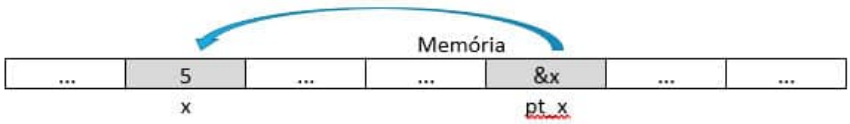
Para entendermos melhor, observe o Exemplo 2. Neste trecho de código, temos a definição de uma variável inteira e de um ponteiro para o tipo de dados inteiro.

```
int x = 5;
int *pt_x;

/* Ponteiro pt_x recebe o endereço de memória da variável x */
pt_x = &x;
```

Exemplo 2: Trecho de código.

A Figura 5 mostra a indireção do exemplo.



Fonte:O Autor

Figura 5: Indireção simples. Fonte: O Autor.

O ponteiro “pt_x” recebe o endereço de memória da variável “x” pela instrução:

pt_x = &x;

Agora, o ponteiro pt_x terá armazenado o endereço de memória da variável x. Portanto, manusear o ponteiro pt_x estará manipulando, indiretamente, a variável x.

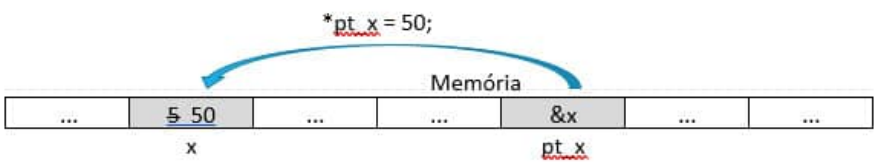
Para acessar o conteúdo do endereço armazenado no ponteiro, basta utilizar o operador ‘*’ antes do nome do ponteiro.

*pt_x

Agora, considere a instrução apresentada abaixo. Note que o valor da variável “x” é alterado pelo ponteiro “pt_x”.

*pt_x = 50;

Como visto na Figura 6, podemos observar que o conteúdo da variável “x” foi alterado de forma indireta, ou seja, não foi feita referência ao nome da variável “x”. Neste caso, o ponteiro é chamado de ponteiro variável, pois assim é possível armazenar qualquer endereço.



Fonte:O Autor

Figura 6: Alteração do conteúdo a partir do ponteiro. Fonte: O Autor.

INDIREÇÃO MÚLTIPLA

O endereço do ponteiro pode ser obtido da seguinte forma:

&pt_x

Sendo assim, um ponteiro pode armazenar o endereço de outro ponteiro, ocasionando uma indireção múltipla.

Para entendermos melhor, vamos analisar a declaração de um ponteiro de indireção múltipla. Utiliza-se N vezes o operador *, sendo N o nível de indireção.

No Exemplo 3, é declarado um ponteiro para ponteiro.

```
int ** pt2; /*ponteiro para ponteiro do tipo inteiro*/
int * pt1; /*ponteiro para o tipo inteiro*/
int x = 10;

pt2 = &pt1;
pt1 = &x;

*pt1 = 30;
**pt2 = 50;
```

Exemplo 3: Declarando um ponteiro para ponteiro.

Sendo:

&pt2: Endereço do ponteiro ‘pt2’;

&pt1: Endereço do ponteiro ‘pt1’;

pt2: Conteúdo de ‘pt2’, ou seja, o endereço de ‘pt1’, que foi recebido através do comando pt2 = &pt1;

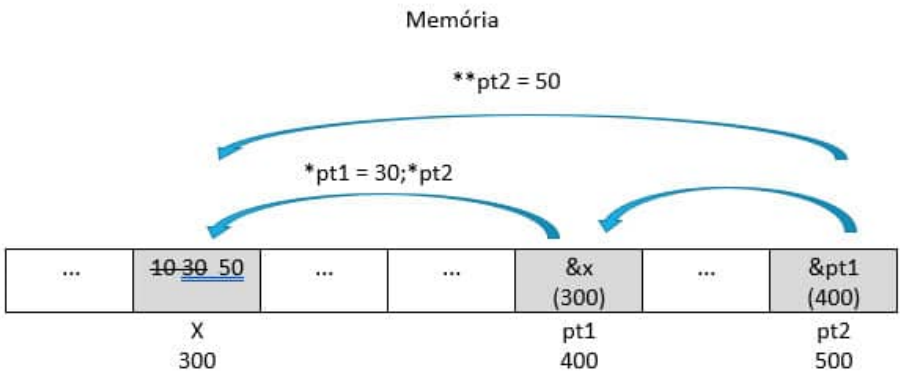
pt1: Conteúdo de ‘pt1’, ou seja, o endereço de ‘x’, que foi recebido através do comando pt1 = &x;

*pt2: Conteúdo do endereço apontado, ou seja, o conteúdo de ‘pt1’.

*pt1: Conteúdo do endereço apontado, ou seja, o conteúdo de ‘x’.

**pt2: Acessa o conteúdo do endereço armazenado no ponteiro que é referenciado por ‘pt2’, ou seja, acessa ‘x’ indiretamente.

A Figura 7 ilustra as operações com ponteiro.



Fonte:O Autor

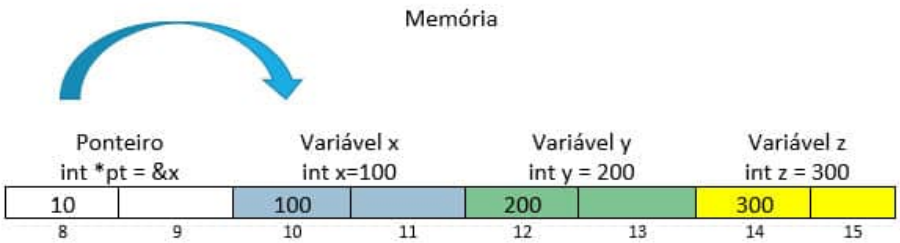
Figura 7: Indireção múltipla fonte: O Autor.

ARITMÉTICA DE PONTEIROS

Os ponteiros permitem apenas as operações de adição e subtração. As operações de soma, subtração e comparação são válidas desde que os ponteiros envolvidos apontem para o mesmo tipo de dado.

No caso das operações básicas de incremento e decremento, analisaremos o comportamento e o que ocorre com o endereço armazenado no ponteiro quando uma das operações aritméticas é utilizada.

Na Figura 8, podemos observar um ponteiro e três variáveis do tipo inteiro.



Fonte:O Autor

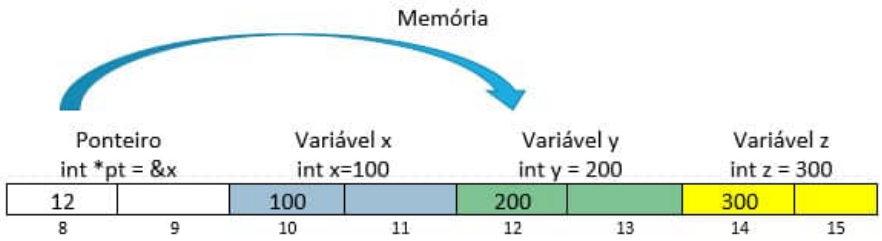
Figura 8: Ponteiro do tipo inteiro apontando para uma variável. Fonte: O Autor.

Consideremos que essas variáveis estão alinhadas sequencialmente na memória e que o ponteiro (pt) está armazenando o endereço da variável “x” (endereço 10).

Para incrementar o ponteiro: pt++;

Ao realizar a operação, o ponteiro passará a apontar para o próximo dado do seu tipo.

Ou seja, ao incrementar o ponteiro, o seu valor é alterado conforme o tamanho do tipo de dado que ele aponta. Diante disso, o próximo valor pt é o endereço 12, logo, o ponteiro aponta para a variável y. Esta situação é ilustrada na Figura 9.



Fonte:O Autor

Figura 9: Operação de incremento. Fonte: O Autor.

Para decrementar o ponteiro: pt--;

Neste caso, o ponteiro passa a apontar para o elemento anterior. Ou seja, ao decrementar o ponteiro (pt), ele irá apontar novamente para a variável x que ocupa a posição 10.

De maneira geral, qualquer operação aritmética de ponteiro será realizada conforme o seu tipo base. Por exemplo:

/* 4 x sizeof(int) = 16 bytes */

pt = pt + 4;

Atenção! Para visualização completa da equação utilize a rolagem horizontal

Como o ponteiro é do tipo inteiro e, para este caso a variável inteiro ocupa 4 bytes, essa operação faz com que o ponteiro seja deslocado 16 bytes (4 x 4 bytes), ou seja, pt irá apontar para quatro elementos adiante.

UTILIZAÇÃO DOS PONTEIROS

Os ponteiros ou apontadores são muito úteis para acessar uma determinada variável em diferentes partes de um programa.

Os ponteiros são úteis em várias situações, por exemplo:

Alocação dinâmica de memória.

Manipulação de arrays.

Para retornar mais de um valor em uma função.

Referência para listas, pilhas, árvores e grafos.

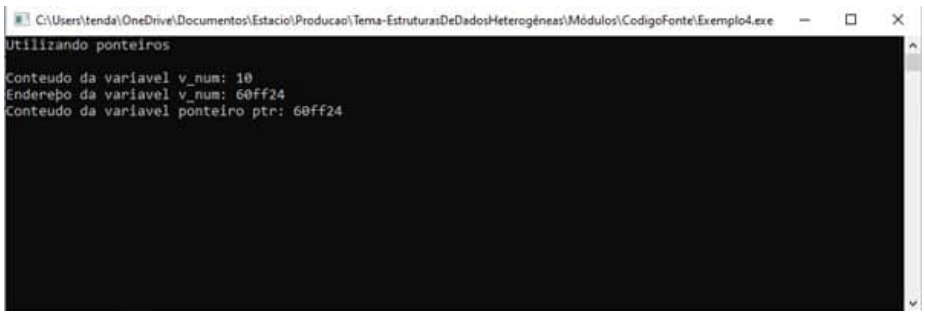
EXEMPLO DE UTILIZAÇÃO DE PONTEIROS

Observemos o exemplo 4 utilizando ponteiro.

```
1#include < stdio.h >
2#include < conio.h >
```

```
3int main(void)
4{
5//v_num é a variável que
6//será apontada pelo ponteiro
7int v_num = 10;
8
9//declaração de variável ponteiro
10int *ptr;
11
12//atribuindo o endereço da variável v_num ao ponteiro
13ptr = &v_num;
14
15printf("Utilizando ponteiros\n\n");
16printf("Conteúdo da variável v_num: %d\n", v_num);
17printf("Endereço da variável v_num: %x \n", &v_num);
18printf("Conteúdo da variável ponteiro ptr: %x", ptr);
19
20getch();
21return(0);
22}
```

Exemplo 4: Utilizando ponteiro.



Fonte:O Autor

Figura 10: Saída da execução do Exemplo 4. Fonte: O Autor.

O Exemplo 4 declara duas variáveis, a variável “v_num” do tipo int e a variável “ptr” do tipo int sendo um ponteiro, de acordo com as linhas 7 e 10 do código-fonte.

```
//v_num é a variável que será apontada pelo ponteiro
int v_num = 10;

//declaração de variável ponteiro
int *ptr;
```

Note que foi usado o operador * para designar que a variável “ptr” é um ponteiro. Como a intenção é armazenar o endereço da variável denominada “v_num”, que é uma variável do tipo int, o ponteiro também tem que ser do tipo int. Isto significa que vai apontar para uma variável do tipo inteiro.

Na linha 13, a variável “ptr” recebe o endereço de memória da variável “v_num”.

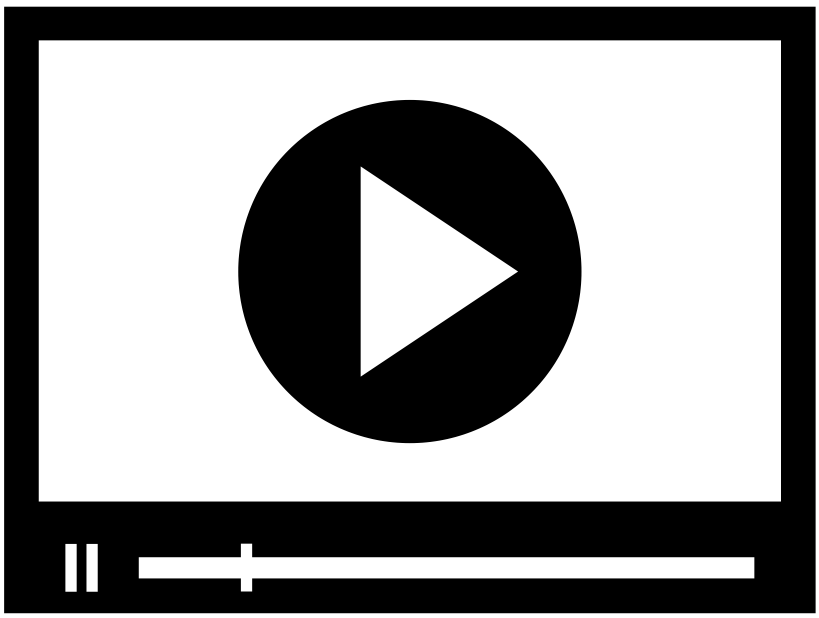
```
//atribuindo o endereço de v_num ao ponteiro
ptr = &v_num;
```

Para atribuir o endereço da variável “v_num” ao ponteiro “ptr”, utilizamos o operador de endereço &, pois estamos nos referindo ao endereço da variável “v_num” e não ao conteúdo.

Nas linhas 15 a 18, será impresso o valor da variável “v_num”, o endereço de memória da variável “v_num” e o valor do ponteiro “ptr”.

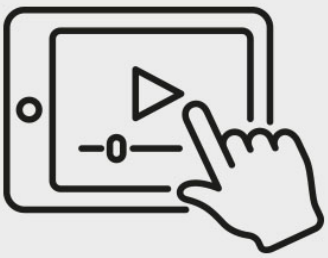
```
printf("Utilizando ponteiros\n\n");
printf("Conteudo da variavel v_num: %d\n", v_num);
printf("Endereço da variavel v_num: %x \n", &v_num);
printf("Conteudo da variavel ponteiro ptr: %x", ptr);
```

Foi utilizado %x para exibir o endereço e o conteúdo do ponteiro ptr, pois trata-se de um valor hexadecimal por ser endereço de memória.



EMPREGANDO PONTEIROS E ALOCAÇÃO DINÂMICA

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



VERIFICANDO O APRENDIZADO

1. A MEMÓRIA É UM COMPONENTE DO COMPUTADOR RESPONSÁVEL PELO ARMAZENAMENTO DE DADOS E INSTRUÇÕES. ELA É COMPOSTA POR PALAVRAS, SENDO CADA PALAVRA IDENTIFICADA UNICAMENTE A PARTIR DE UM ENDEREÇO, OU SEJA, UM ENDEREÇO DE MEMÓRIA. A ALOCAÇÃO DE MEMÓRIA NA LINGUAGEM C PODE SER DE TRÊS TIPOS:

A) Elástica, manual e estática

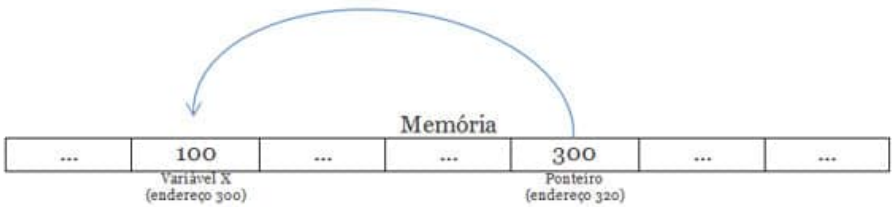
B) Elástica, automática e dinâmica

C) Estática, manual e estática

D) Elástica, manual e dinâmica

E) Estática, automática e dinâmica

2. SABEMOS QUE TODO DADO ARMAZENADO EM MEMÓRIA POSSUI UM ENDEREÇO E QUE A DEFINIÇÃO DE UM PONTEIRO É UMA VARIÁVEL QUE GUARDA O ENDEREÇO DE MEMÓRIA, OU SEJA, A LOCALIZAÇÃO DO DADO. DIANTE DISTO, A FIGURA APRESENTA UMA MEMÓRIA QUE ARMAZENA A VARIÁVEL "X" E O PONTEIRO PARA A VARIÁVEL "X". COMO O CONTEÚDO DO PONTEIRO É UM ENDEREÇO, A SETA INDICA A RELAÇÃO ENTRE O PONTEIRO E A VARIÁVEL QUE ELE APONTA. QUAL O VALOR DA VARIÁVEL "X" E ESTE DADO ESTÁ ARMAZENADO EM QUAL ENDEREÇO DE MEMÓRIA, ISTO É, QUAL É O VALOR ARMAZENADO PELO PONTEIRO "PT_X"?



A) x= 300 e pt_x = 300

B) x=300 e pt_x=100

C) x=300 e pt_x=320

D) x=100 e pt_x=320

E) x=100 e pt_x=300

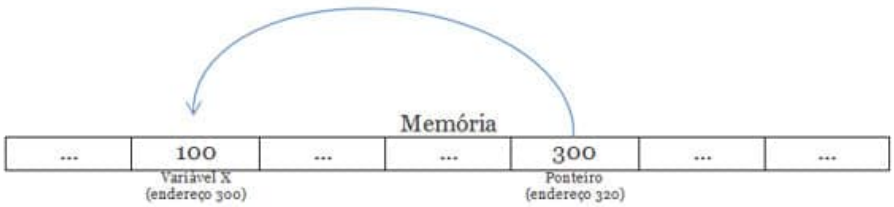
GABARITO

1. A memória é um componente do computador responsável pelo armazenamento de dados e instruções. Ela é composta por palavras, sendo cada palavra identificada unicamente a partir de um endereço, ou seja, um endereço de memória. A alocação de memória na linguagem C pode ser de três tipos:

A alternativa **"E "** está correta.

A alocação de memória na linguagem C pode ser de três tipos: Alocação estática, alocação automática e alocação dinâmica.

2. Sabemos que todo dado armazenado em memória possui um endereço e que a definição de um ponteiro é uma variável que guarda o endereço de memória, ou seja, a localização do dado. Diante disto, a figura apresenta uma memória que armazena a variável "x" e o ponteiro para a variável "x". Como o conteúdo do ponteiro é um endereço, a seta indica a relação entre o ponteiro e a variável que ele aponta. Qual o valor da variável "x" e este dado está armazenado em qual endereço de memória, isto é, qual é o valor armazenado pelo ponteiro "pt_x"?



A alternativa **"E "** está correta.

Na figura, o valor da variável "x" é 100 e esse dado está armazenado no endereço de memória 300. Como o ponteiro também ocupa espaço, o endereço de memória da variável "x" está armazenado no endereço 320 da memória. Sendo assim, o valor da variável "x" é 100 e o valor do ponteiro pt_x é 300 (endereço de memória da variável "x").

MÓDULO 2

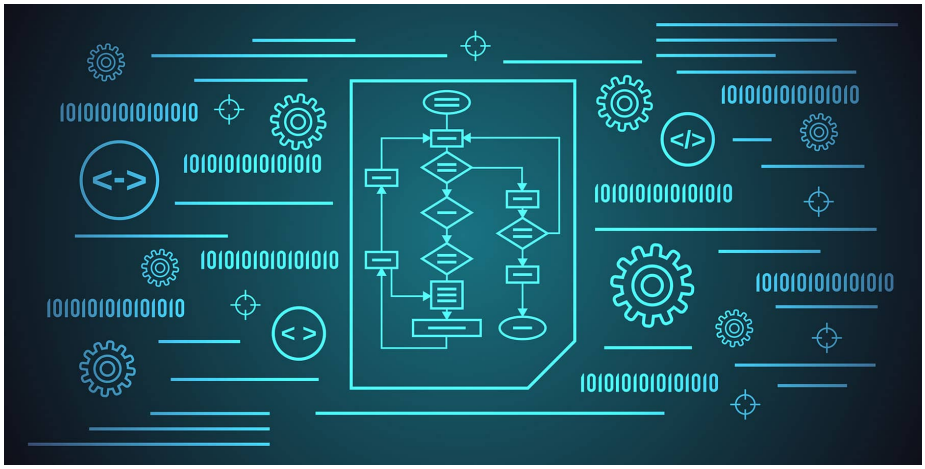
⦿ **Definir estrutura de dados heterogênea**

ESTRUTURA DE DADOS

Estrutura de dados é o ramo da Computação que estuda os diversos mecanismos de organização de dados para atender aos diferentes requisitos de processamento.

Sendo assim, podemos entender estrutura de dados como um modo de armazenar e organizar os dados no computador, de uma maneira que eles possam ser usados eficientemente.

As estruturas de dados definem a organização, métodos de acesso e opções de processamento para coleções de itens de informação manipulados pelo programa. As estruturas são formas de distribuir e relacionar os dados disponíveis, de modo a tornar mais eficientes os algoritmos que manipulam esses dados.



Fonte: Shutterstock

MAS O QUE É ALGORITMO?

Algoritmo é um conjunto de passos finitos e organizados, que são usados para executar uma tarefa específica. Funciona como um bloco de informações que permite que dispositivos como computadores funcionem. E o programa é um código feito de algoritmos que dizem o que deve ser realizado.

Os algoritmos manipulam os dados. Quando os dados estão dispostos de forma coerente, caracterizam uma forma, uma estrutura de dados.

Geralmente, os algoritmos são elaborados para manipulação de dados e quando os dados estão organizados de forma coerente, representam uma estrutura de dados. Quando um programador cria o algoritmo para solucionar um problema, ele também cria uma estrutura de dados que é manipulada pelo algoritmo.

TIPOS DE ESTRUTURA DE DADOS

Uma estrutura de dados pode ser dividida em: **Dado e estrutura**.

O **dado** é o elemento que possui valor agregado e que pode ser utilizado para solucionar problemas computacionais. Os dados possuem tipos específicos. Por padrão, são quatro tipos de dados.

Inteiro	Representa valores numéricos negativos ou positivos sem casa decimal, ou seja, valores inteiros.
Real	Representa valores numéricos negativos ou positivos com casa decimal, ou seja, valores reais. Também são chamados de ponto flutuante.
Lógico	Representa valores booleanos, assumindo apenas dois estados, verdadeiro ou falso. Pode ser representado apenas um bit (que aceita apenas 1 ou 0).

Texto	Representa uma sequência de um ou mais caracteres, colocamos os valores do tipo texto entre “” (aspa duplas).

Atenção! Para visualização completa da tabela utilize a rolagem horizontal

O tipo de dados texto pode ser subdividido em string ou caractere, e o tipo de dado real em ponto flutuante com precisão simples (float) ou ponto flutuante com precisão dupla (double).

A estrutura é o elemento responsável por carregar as informações dentro de uma estrutura de software. Alguns tipos de estrutura são: Vetores multidimensionais, pilhas, filas, listas, árvores, grafos, tabelas hashing, dentre outros.

Vamos conhecer os principais tipos de estrutura de dados citados anteriormente.

Vetores unidimensionais e bidimensionais	Fila	Árvore
Lista encadeada	Pilha	Grafo

Atenção! Para visualizaçãocompleta da tabela utilize a rolagem horizontal

VETORES UNIDIMENSIONAIS E BIDIMENSIONAIS

Também conhecidos por arrays, são algumas das estruturas de dados mais simples e mais utilizadas dentre todas. Os dados armazenados neste tipo de estrutura são do mesmo tipo, portanto, dizemos que é uma estrutura de dados homogênea. Nesta estrutura, a adição e pesquisa de novos elementos é realizada de forma aleatória e o acesso aos elementos é através de índices. No caso dos vetores unidimensionais, só existe uma linha, e no caso dos bidimensionais, são empregadas linhas e colunas, formando uma matriz.

FILA

É uma estrutura de dados com a ideia de organizar elementos em fila. Basicamente, uma fila é uma lista que só vai para um lado, ou seja, novos elementos são inseridos no final da fila e os elementos a serem removidos estão no início da fila. É comum utilizar a expressão FIFO (First-In-First-Out) para ilustrar o funcionamento da fila, que significa primeiro a entrar – primeiro a sair.

ÁRVORE

É uma estrutura de dados que caracteriza uma relação de hierarquia entre os elementos. Por exemplo, para se construir um organograma de uma empresa, visualizando o conjunto de pessoas que lá trabalham, tendo em conta sua função, a estrutura de dados árvore é que mais se aplica. Na estrutura da empresa, uma pessoa não pode pertencer a dois departamentos diferentes, cada departamento tem a sua própria diretoria.

LISTA ENCADEADA

É uma estrutura de dados que implementa um conjunto de dados ordenados e do mesmo tipo, ou seja, é uma sequência de zero ou mais elementos de um determinado tipo. Uma diferença básica entre os vetores e a lista é que nesta os elementos não precisam estar fisicamente em sequência, ou seja, eles podem ocupar posições de memória não sequenciais.

PILHA

É uma estrutura de dados amplamente utilizada e que implementa a ideia de pilha de elementos. Basicamente, é uma sequência de elementos dispostos “um em cima do outro”, mas com uma regra para entrada e saída dos elementos: O último elemento que chega é o primeiro que sai da estrutura. É comum utilizarmos a expressão LIFO (Last-In-First-Out) para designar a pilha, que significa último a entrar – primeiro a sair. Para que o acesso a um elemento da pilha ocorra, os demais acima devem ser removidos.

GRAFO

É uma estrutura de dados bastante genérica, que organiza vários elementos, estabelecendo relações entre eles, dois a dois. Por exemplo, ao se estabelecer um trajeto para percorrer todas as capitais do NE do Brasil, devemos utilizar um mapa que indique as rodovias existentes e estabeleça uma ordem para percorrer as cidades. Para este caso, a estrutura de dados grafo é mais apropriada.

ESTRUTURAS DE DADOS HOMOGÊNEAS X HETEROGÊNEAS

As estruturas de dados também se dividem em homogêneas e heterogêneas.

As **estruturas de dados homogêneas** são conjuntos de dados formados pelo mesmo tipo de dados. Esta estrutura permite o agrupamento de várias informações ou valores dentro de uma mesma variável. Como exemplo, podemos citar os vetores e as matrizes.

Um vetor é um arranjo de elementos que armazenamos na memória principal, com posições contíguas de memória, todos com o mesmo nome. Sendo assim, podemos entender que os vetores são estruturas lineares e estáticas, isto é, são compostas por um número finito e predeterminado de valores.

Por exemplo, na Figura 11, é possível observarmos um vetor de notas com 9 posições, onde cada posição corresponde a um índice do vetor.

Nas estruturas de dados homogêneas, a individualização de cada variável é feita através do uso de índices. No caso dos vetores que são matrizes de uma só dimensão, é necessário apenas 1 índice para acesso às variáveis.

Notas:	6,1	2,3	9,4	5,1	8,9	9,8	10	7,0	6,3	4,4
Posição:	0	1	2	3	4	5	6	7	8	9

Fonte: O Autor

Figura 11: Exemplo de vetor. Fonte: O Autor.

Na Figura 12, temos um exemplo de matriz. As matrizes possuem mais de uma dimensão, isto é, necessitam de um índice para cada dimensão para acesso às variáveis. E assim como nos vetores, cada posição dentro de uma matriz possui um índice.

Prateleira \ Posição do livro	0	1	2	...	n-1
0	788	598	265	...	156
1	145	258	369	...	196
2	989	565	345	...	526
⋮	⋮	⋮	⋮	⋱	⋮
m-1	845	153	564	892	210

Fonte:O Autor

Figura 12: Exemplo de matriz. Fonte: O Autor.

As **estruturas de dados heterogêneas** são conjuntos de dados formados por tipos de dados diferentes, como os registros. Para entendermos melhor, citemos como exemplo uma agenda telefônica, na qual teremos informações de vários tipos: Nome, telefone, endereço etc.

Uma das principais estruturas de dados heterogêneas é o registro. Um registro pode ser composto de vários campos, componentes ou elementos e cada um deles pode ser de um tipo diferente.

Chamaremos cada elemento de um registro de campo, e para cada campo teremos que definir o seu tipo e o seu identificador. É possível definir quantos campos forem necessários.

Por exemplo, imaginamos uma ficha de cadastro com os seguintes componentes: Matrícula, nome, data de nascimento, cargo e salário. Para este caso, utilizamos o registro com os campos: Matrícula, nome, data de nascimento, cargo e salário. Estes campos do registro, que são de tipos diferentes, são partes que especificam cada uma das informações, conforme mostra a Tabela 3:

Registro Funcionário	
Matrícula	Tipo Inteiro
Nome	Tipo Cadeia de Caracteres
Dt. Nascimento	Tipo Data
Cargo	Tipo Cadeia de Caracteres
Salário	Tipo Real

Atenção! Para visualização completa da tabela utilize a rolagem horizontal

A definição dos campos de um registro é similar à declaração de variáveis. Portanto, assim como na declaração de variáveis, podemos declarar campos do mesmo tipo, na mesma linha, separados por vírgulas.

A declaração de um registro segue a seguinte estrutura logo abaixo, onde cada campo funciona como uma variável primitiva, mas todos os campos estão agrupados no registro.

```
<nome_da_variavel>:registro
Inicio
    <nome_campo_1>:<tipo_campo_1>
    <nome_campo_2>:<tipo_campo_2>
    ...
    <nome_campo_n>:<tipo_campo_n>
Fimregistro
```

Abaixo, temos o exemplo de declaração da estrutura de dados heterogênea (registro) para a ficha de cadastro dos funcionários de uma empresa.

```
var Ficha_Funcionario: registro
```

```
inicio
    matricula: inteiro
    nome: vetor[1..50] de caractere
    Dt_Nascimento: vetor[1..9] de caractere
    Cargo: vetor[1..30] de caractere
    Salario: real
fim
```

A variável Ficha_Funcionario agrupa as informações referentes a um funcionário da empresa. Esta estrutura constitui um recurso importante para organizar os dados utilizados por um programa, pois trata um grupo de valores como uma única variável.

Para acessar este tipo de variável, é necessário especificar o registro nominando o campo que se deseja utilizar. Ou seja, para manipularmos um registro, temos que normalmente fazê-lo campo a campo, como no caso dos elementos dos vetores.

A única diferença é que podemos fazer atribuição direta entre variáveis registro, o que não pode ser feito entre vetores. Ao se fazer uma atribuição entre duas variáveis registro, automaticamente o conteúdo de todos os campos de uma variável, mesmo que alguns deles sejam vetores, é copiado para os campos da outra variável.

Sendo assim, para se ter acesso a um campo de uma variável registro, seja para utilizar o valor desse campo ou alterar o seu conteúdo, faremos do seguinte modo:

varReg.campo

Ao voltarmos ao nosso exemplo, para se acessar o campo matricula, nome e cargo, devem-se realizar as seguintes operações:

leia(Ficha_Funcionario.matricula)

escreva(Ficha_Funcionario.nome)

Ficha_Funcionario.***cargo*** <- “gerente”

Os registros são geralmente utilizados quando queremos trabalhar com tabelas heterogêneas ou quando estamos trabalhando com arquivos binários. No caso de se trabalhar com tabelas heterogêneas, criamos uma tabela heterogênea simples criando-se um vetor de registros.

Tanto os vetores como os registros são usados com a finalidade de resolver problemas mais complexos. Ao se considerar que um vetor é um conjunto de elementos do mesmo tipo, é natural que, quando precisamos agrupar vários registros, por exemplo, várias Fichas de Funcionário, utilizemos vetores de registro.

Os vetores de registro são utilizados para armazenar conjuntos de elementos complexos. No caso do exemplo, no lugar de armazenar apenas os dados do Funcionário, um vetor de registro pode armazenar as informações cadastrais de todos os 50 (cinquenta) Funcionários da empresa.

```
Algoritmo exemplo
var funcionario: vetor [1..50] de registro
Inicio
        matricula: inteiro
```

```
        nome: caractere
        dtNascimento: caractere
        cargo: caractere
        salario: real
fimregistro
```

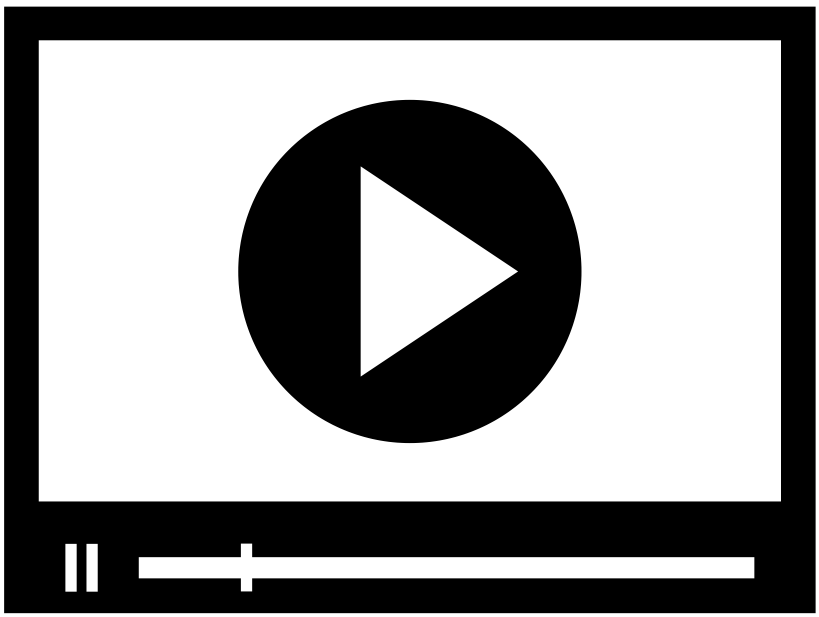
Para cada um dos elementos que constituem um registro, é dado o nome do campo. Sendo assim, no exemplo acima, temos os campos: Matrícula, nome, data de nascimento, cargo e salário.

Ao analisarmos melhor o exemplo, podemos pensar neste vetor de registros como sendo uma tabela de 50 linhas e 5 colunas, onde a primeira coluna seria para guardar valores de inteiro simples, a segunda, terceira e quarta colunas para guardar valores strings e a quinta coluna para guardar valores de inteiro longo, conforme é ilustrado na Tabela 4.

Matrícula	Nome	D. Nascimento	Cargo	Salário

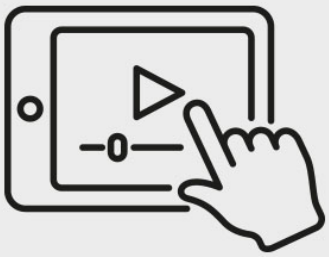
Tabela 4 : Exemplo da Tabela de Registro de Funcionário.

Atenção! Para visualização completa da tabela utilize a rolagem horizontal



ESTRUTURA DE DADOS HOMOGÊNEAS E HETEROGÊNEAS

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



VERIFICANDO O APRENDIZADO

1. ESTUDAMOS SOBRE OS TIPOS DE ESTRUTURA DE DADOS. A ESTRUTURA DE DADOS EM QUE O PRIMEIRO ELEMENTO INSERIDO É O PRIMEIRO ELEMENTO A SER RETIRADO É DENOMINADA:

A) Pilha

B) Matriz

C) Árvore binária

D) Fila

E) Lista

2. AS ESTRUTURAS DE DADOS HETEROGÊNEAS SÃO CONJUNTOS DE DADOS FORMADOS POR TIPOS DE DADOS DIFERENTES. SENDO ASSIM, SELECIONE UMA DAS PRINCIPAIS ESTRUTURAS DE DADOS HETEROGÊNEAS.

A) Pilha

B) Matriz

C) Árvore binária

D) Registro

E) Lista

GABARITO

1. Estudamos sobre os tipos de estrutura de dados. A estrutura de dados em que o primeiro elemento inserido é o primeiro elemento a ser retirado é denominada:

A alternativa **"D "** está correta.

A estrutura de dados fila são listas em que as operações de remoção e inserção ocorrem sempre em locais específicos. A inserção é feita sempre no final da lista e a remoção sempre no início da lista. Em função disto, uma fila assume a condição FIFO.

2. As estruturas de dados heterogêneas são conjuntos de dados formados por tipos de dados diferentes. Sendo assim, selecione uma das principais estruturas de dados heterogêneas.

A alternativa **"D "** está correta.

As estruturas de dados heterogêneas são conjuntos de dados formados por tipos de dados diferentes, como os registros. Os registros são úteis para os casos em que é necessário guardar informações de diversos tipos, porém, como se fossem uma única variável.

MÓDULO 3

⦿ **Aplicar structs com a utilização da linguagem de programação C**

DEFINIÇÃO DE STRUCT EM C

Na linguagem C, existem dois tipos de dados: Os tipos básicos (int, char, float, double, void) e os tipos compostos homogêneos (arrays). Porém, nem sempre estes tipos são suficientes para o programa e, por isso, a linguagem C nos permite a criação de outras estruturas de dados, a partir dos tipos básicos, como os registros.

Na linguagem C, os registros são representados com uma estrutura denominada struct, que é uma coleção de variáveis relacionadas, usando um nome comum. As estruturas de dados em linguagem C podem conter variáveis de tipos diferentes de dados, ao contrário de um array, que só pode conter dados de um mesmo tipo.

❓ VOCÊ SABIA

As estruturas (struct) são usadas para definir registros que são armazenados em arquivos e, com os ponteiros, simplificam a criação de estruturas de dados mais complexas, por exemplo, pilhas, listas ligadas, filas e árvores.

Sendo assim, entendemos que as estruturas são um tipo de dados derivado, o que significa que são construídas com o uso de objetos de outros tipos existentes.

Definimos um modelo de estrutura na linguagem C como mostrado na figura.

```
struct identificador {
    tipo variável; tipo variável;
```

```
    tipo variável;
    tipo variável;
};
```

Em resumo, struct é um conjunto de variáveis sob o mesmo nome, e definido pelo programador. E cada variável dentro de uma estrutura pode ser de um tipo diferente, ou melhor, não é obrigatório que sejam todas do mesmo tipo.

Visualizamos, logo abaixo, um exemplo de definição de struct. A palavra-chave struct inicia o bloco de definição da estrutura. Logo após, vem o nome que desejamos atribuir à estrutura (tag da estrutura), e então são abertas chaves "**{**" para iniciar a seção do corpo da estrutura.

```
struct mystruct {
  char a;
  int b;
  flaot c;
};
```

Neste outro exemplo, é possível visualizar a definição de uma estrutura de nome endereço, contendo os membros rua, número, CEP e bairro:

```
struct endereco {
    char rua[50];
    char numero[5];
    char CEP[8];
    char bairro[30];
};
```

DECLARAÇÃO DE STRUCT EM C

A declaração de estrutura de dados struct é realizada da seguinte forma:

```
struct nome_da_estrutura {
    tipo_campo1  nome_campo1;
    tipo_campo2  nome_campo2;
    ...
} variáveis_que_armazenam_a_estrutura;
```

Onde:

Struct é a palavra-chave para estrutura.

Tag da estrutura é a palavra nome_da_estrutura. A tag de estrutura é usada para declarar variáveis deste tipo de estrutura, é o que identifica este tipo de estrutura.

Membros ou campos são as variáveis dentro da estrutura.

Variáveis_que_armazenam_a_estrutura é o nome da variável que irá armazenar os valões da struct.

No corpo da estrutura, são definidas as variáveis ou membros do tipo struct, cada uma com seu nome, sem repetições, e podem ser de qualquer tipo básico de dados (int, float, char etc.), ou ainda, tipos de dados agregados, como arrays ou até mesmo outras estruturas – desde que não contenha uma instância de si mesma. É comum chamar os membros de elementos ou campos.

Note que nenhuma variável é realmente declarada com este comando – apenas definimos o formato dos dados da estrutura. Basicamente, um tipo complexo de variável. Se quisermos declarar uma variável, podemos fazê-lo ao final da estrutura (**variáveis_que_armazenam_a_estrutura;**) ou com as declarações de outras variáveis, como veremos a seguir.

A declaração da estrutura de dados struct pode ser feita de três maneiras diferentes.

A **primeira** é a declaração do tipo struct realizada da mesma forma que declaramos variáveis de outros tipos quaisquer. Por exemplo, vamos declarar a variável x para que seja do tipo struct endereço com a declaração a seguir:

struct endereco x;

A **segunda** maneira é a declaração de uma ou mais variáveis durante a definição da própria estrutura. Por exemplo, podemos declarar três variáveis de nomes livro1, livro2 e livro3 a partir de uma estrutura de nome livro.

```
struct livro {
    char nome[30];
    char autor[50];
    int paginas;
    float preco;
} livro1, livro2, livro3;
```

E por último, a **terceira** maneira é para o caso de apenas uma variável do tipo estrutura ser necessária no programa. Sendo assim, não precisamos declarar um identificador, basta definir o nome da variável que será criada. Por exemplo, podemos criar uma variável estrutura chamada livro da seguinte forma:

```
struct {
    char nome[30];
```

```
    char autor[50];
    int paginas;
    float preco;
} livro;
```

Como pode ser visto, nesta última forma de declaração, o nome para a tag de estrutura é opcional. Entretanto, se não for declarado um nome, esta última forma de declaração é a única maneira de declarar as variáveis do tipo estrutura. Apesar de ser possível não dar um nome à estrutura, sempre forneça um, pois é uma prática recomendada.

Podem ser feitas diversas operações com as structs em linguagem C, como:

Atribuição de variáveis da estrutura a variáveis da estrutura do mesmo tipo.

Leitura do endereço de uma variável de estrutura (operador &).

Acesso aos membros de uma variável de estrutura.

Uso do operador sizeof para determinar o tamanho de uma variável de estrutura.

Veremos a seguir algumas dessas operações.

INICIALIZANDO STRUCT EM C

A estrutura de dados struct em linguagem C possui uma inicialização parecida com os vetores e as matrizes.

Uma das maneiras de inicializar uma struct é através de uma lista de inicialização, como um array. Agora, se o número de inicializadores na lista for menor que os membros na estrutura, os membros restantes serão automaticamente inicializados em zero ou Null, se o membro for um ponteiro.

Por exemplo, podemos inicializar a estrutura endereço usando a declaração a seguir. Observe que os elementos do array são passados aos membros da estrutura na ordem em que foram declarados.

struct endereco x = {"Av. das Américas", "4200", " 22640-102 ", "Barra da Tijuca"};

Outra forma é atribuindo uma variável estrutura do mesmo tipo, já inicializada.

Por último, atribuindo valores aos membros individuais da estrutura. Por exemplo, suponha que desejamos atribuir o valor “Av. das Américas” ao membro rua da estrutura endereco.

Podemos usar a declaração a seguir, referenciando o nome da variável estrutura, seguida por um ponto e pelo nome do membro (campo) que receberá a atribuição do dado.

endereco.rua = "Av. das Américas";

ACESSANDO OS MEMBROS DA STRUCT EM C

Para acessar os membros de uma estrutura de dados struct, podem ser utilizados dois tipos de operadores:

Operador de membro de estrutura . (operador de ponto ou de seleção direta).

Operador de ponteiro de estrutura -> (operador de seta).

Quando é declarada uma variável do tipo estrutura, acessamos um membro da estrutura usando o operador ponto, indicando o nome da estrutura, seguido por um ponto e pelo nome do campo que se quer acessar. Neste caso, dizemos que a estrutura é diretamente referenciada.

Por exemplo, podemos acessar e visualizar o campo rua da estrutura x, que é uma variável da struct endereço,) usando a seguinte declaração:

printf("%s", x.rua);

Quando a struct for referenciada através de ponteiros, o emprego é semelhante ao operador ponto, sendo este substituído pelo operador seta **(->)**.

Por exemplo, suponha que um ponteiro xptr tenha sido declarado e aponte para a estrutura endereco, e que o endereço de uma estrutura x tenha sido atribuído ao ponteiro xprt. Para acessar o membro rua da estrutura endereco via operador de ponteiro, usamos a declaração a seguir:

printf("%s", x->rua);

MANIPULANDO STRUCTS

As estruturas de dados structs podem ser manipuladas com o objetivo de acessar os campos, atribuir valores para os campos, imprimir os valores dos campos, dentre outros.

Para atribuir valores aos campos da estrutura, você faz isto diretamente, e em qualquer parte do programa, conforme a seguir:

```
aluno_especial.codigo = 10;
strcpy(aluno_especial.nome, "Manoel");
aluno_especial.nota = 10.0;
```

Para atribuir um valor a uma string, é necessário utilizar a função strcpy (CPY = copiar; STR = string). A função copiará o que está dentro das aspas duplas para o campo STRING da estrutura.

É possível imprimir os valores dos campos da estrutura em qualquer parte do programa utilizando a função printf.

```
printf(" \n %d ", aluno_especial.codigo);
printf(" \n %s ", aluno_especial.nome);
```

```
printf(" \n %.2f ", aluno_especial.nota);
```

Entretanto, se for preciso imprimir todos os membros da estrutura de uma única vez, é recomendável criar uma função para isto.

```
void imprimir(Aluno aluno_regular){
	printf(" \n %d ", aluno_especial.codigo);
	printf(" \n %s ", aluno_especial.nome);
	printf(" \n %.2f ", aluno_especial.nota);
}
```

Observe que a função de impressão é do tipo Void, mas tem um argumento, isto é, é definida uma variável do tipo da estrutura, significando que, ao chamar a função, você deverá passar como parâmetro a estrutura que está trabalhando.

A chamada fica da seguinte forma:

```
imprimir(aluno_especial);
```

Para obter dados do teclado, devemos utilizar a função scanf, como em outras situações para entrada de dados na linguagem C. Podemos obter dados do teclado em qualquer parte do programa, assim como também podemos definir uma função para realizar este trabalho.

```
printf(" Digite o código do aluno especial: ");
scanf("%d%*c", &aluno_especial.codigo);
printf(" Digite o nome do aluno especial: ");
scanf("%s%*c", &aluno_especial.nome);
printf(" Digite a nota do aluno especial: ");
scanf("%f%*c", &aluno_especial.nota);
```

```
void cadastrar(Aluno aluno_especial){
	printf(" Digite o código do aluno especial: ");
	scanf("%d%*c", &aluno_especial.codigo);
	printf(" Digite o nome do aluno especial: ");
	scanf("%s%*c", &aluno_especial.nome);
	printf(" Digite a nota do aluno especial: ");
	scanf("%f%*c", &aluno_especial.nota);
}
```

Observe que na função cadastrar e imprimir é passado como parâmetro (por valor) a variável do tipo estrutura. Em algumas ocasiões, você precisará fazer desta forma, em outras, talvez você não tenha que passar a estrutura como parâmetro.

EXEMPLOS PRÁTICOS

No Exemplo 5, vamos criar uma struct para armazenar os dados do aluno.

Exemplo 5:

```
1/* Cria uma estrutura para armazenar dados de um aluno*/
2#include < stdio.h >
3 #include < stdlib.h >
4
5 struct aluno {
6int v_nmat; //número da matrícula
7float v_nota[3]; //notas
8float v_media; //media
9};
10
11int main() {
12struct aluno Felipe; //declara uma variável do tipo struct
13Felipe.v_nmat = 120;
14Felipe.v_nota[0]=8.5;
15Felipe.v_nota[1]=7.2;
16Felipe.v_nota[2]=5.4;
17Felipe.v_media=(Felipe.v_nota[0]+ Felipe.v_nota[1]+ Felipe.v_nota[2])/3.0;
18printf("Matricula:/d\n", Felipe.v_nmat);
19printf("Media: %2f\n", Felipe.v_media);
20system("pause");
21return(0);
22}
```

Neste Exemplo 5, é declarada a struct aluno, conforme a linha 5 do código-fonte.

A struct aluno é composta pelos campos matrícula, nota e médica, onde o nome do campo matrícula é v_nmat do tipo int, o nome do campo nota é v_nota do tipo float e o nome do campo média é v_media do tipo float. O campo nota é também um vetor de 3 posições, conforme sua declaração na linha 7.

Na linha 12, é declarada a variável Felipe do tipo struct aluno.

A struct aluno Felipe recebe os valores para os seus campos nas linhas 13 a 17. Na linha 13, o campo v_nmat recebe o valor de 120. Nas linhas 14, 15 e 16, o campo v_nota, que é um array, recebe os valores para as três posições do vetor.

Na linha 17, para o último campo da struct aluno é calculada a média do aluno e armazenada.

O exemplo finaliza com a impressão da matrícula (linha 18) e da média (linha 19) das notas do aluno Felipe.

Após a execução do programa, temos o retorno com o valor das variáveis ilustradas na Figura 13.

Fonte:Shutterstock

Figura 13: Saída da execução do Exemplo 5. Fonte: O Autor.

Vamos analisar mais um exemplo.

No caso do Exemplo 6, criamos a struct ficha_de_aluno. Depois de criar a struct, precisamos criar a variável que vai utilizá-la. Para tanto, criamos a variável aluno, que será do tipo ficha_de_aluno.

Exemplo 6

```
1 /* Ficha de Aluno */
2 #include < stdio.h >
3 #include < conio.h >
4 int main(void)
5 {
6 /*Criando a struct */
7 struct ficha_de_aluno
8 {
9 char nome[50];
10 char disciplina[30];
11 float nota_prova1;
12 float nota_prova2;
13 };
14
15 /*Criando a variável aluno que será do
16 tipo struct ficha_de_aluno */
17 struct ficha_de_aluno aluno;
18
19 printf("\n---------- Cadastro de aluno -----------\n\n\n");
20 printf("Nome do aluno .....: ");
21 fflush(stdin);
22 fgets(aluno.nome, 40, stdin);
23 printf("Disciplina .....: ");
24 fflush(stdin);
25 fgets(aluno.disciplina, 40, stdin);
26 printf("Informe a 1a. nota ..: ");
27 scanf("%f", &aluno.nota_prova1);
28 printf("Informe a 2a. nota ..: ");
29 scanf("%f", &aluno.nota_prova2);
30 printf("\n\n --------- Lendo os dados da struct ---------\n\n");
```

```
31 printf("Nome ...........: %s", aluno.nome);
32 printf("Disciplina .....: %s", aluno.disciplina);
33 printf("Nota da Prova 1 ...: %.2f\n" , aluno.nota_prova1);
34 printf("Nota da Prova 2 ...: %.2f\n" , aluno.nota_prova2);
35 getch();
36 return(0);
37 }
```

No Exemplo 6, é declarada a struct ficha_de_aluno composta pelos seguintes campos: Nome, disciplina, nota_prova1 e nota_prova2. O campo nome é uma variável do tipo char e um vetor com 50 itens, o campo disciplina é uma variável do tipo char e um vetor com 30 itens e os campos nota_prova1 e nota_prova2 são variáveis do tipo float.

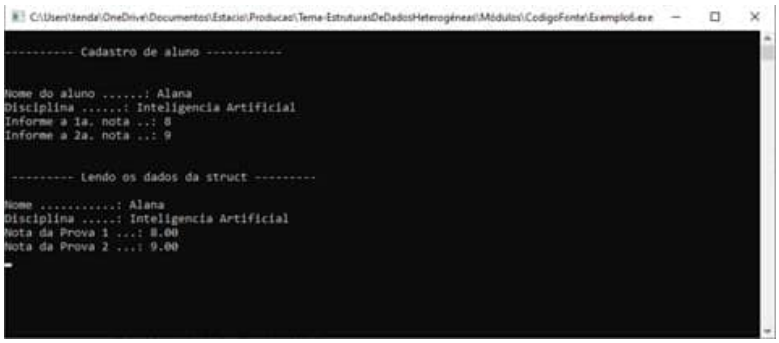
Ao continuar com o reconhecimento do código-fonte do Exemplo 6, é observado o uso da instrução fgets() (linhas 22 e 25) com o objetivo de ler strings, no caso, o nome do aluno e a disciplina.

fgets(variavel, tamanho da string, entrada)

Nesse caso, a entrada é stdin (entrada padrão), pois estamos lendo do teclado, mas, em outro caso, a entrada também poderia ser um arquivo.

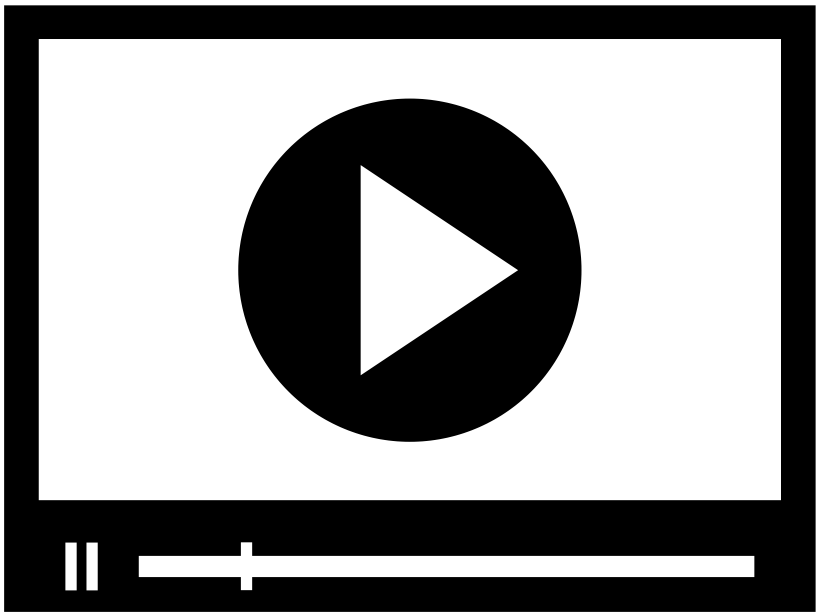
Continuando, a struct ficha_de_aluno aluno recebe os valores para os campos nota_prova1 e nota_prova2 nas linhas 27 e 29.

Após a execução do programa, temos o retorno com o valor das variáveis ilustradas na Figura 14.



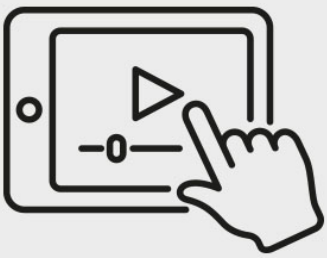
Fonte: O Autor

Figura 14: Saída da execução do Exemplo 6. Fonte: O Autor.



EMPREGANDO *STRUCTS*

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



VERIFICANDO O APRENDIZADO

1. PARA ACESSAR OS MEMBROS DE UMA ESTRUTURA DE DADOS STRUCT, SÃO EMPREGADOS DOIS TIPOS DE OPERADORES: OPERADOR DE MEMBRO DE ESTRUTURA . (OPERADOR DE PONTO) E OPERADOR DE PONTEIRO DE ESTRUTURA -> (OPERADOR DE SETA). SELECIONE A OPÇÃO CORRETA QUE APRESENTA O COMANDO PARA ACESSAR O ELEMENTO RUA DE UMA STRUCT ENDEREÇO QUE FOI REFERENCIADA DIRETAMENTE PELA VARIÁVEL X.

A) printf("%s", x.rua);

B) printf(“%s”, x.->rua);

C) printf(“%s”, x->.rua);

D) printf(“%s”, rua.x);

E) printf(“%s”, rua->x);

2. A ESTRUTURA DE DADOS STRUCTS EM LINGUAGEM C PODE REALIZAR DIVERSAS OPERAÇÕES. DENTRE AS OPÇÕES ABAIXO, IDENTIFIQUE UMA OPERAÇÃO POSSÍVEL A SER REALIZADA COM UMA STRUCT.

A) Acesso aos membros de uma variável de estrutura referenciada diretamente através do operador seta (->).

B) Leitura do endereço de uma variável de estrutura através do operador ponto (.).

C) Acesso aos membros de uma variável de estrutura referenciada diretamente através do operador ponto (.)

D) Uso do operador sizeof para determinar o tempo de acesso para uma estrutura.

E) Uso do operador seta (->) para acessar uma estrutura referenciada diretamente.

GABARITO

1. Para acessar os membros de uma estrutura de dados struct, são empregados dois tipos de operadores: Operador de membro de estrutura . (operador de ponto) e Operador de ponteiro de estrutura -> (operador de seta). Selecione a opção correta que apresenta o comando para acessar o elemento rua de uma struct endereço que foi referenciada diretamente pela variável x.

A alternativa **"A "** está correta.

Para acessar um membro de uma estrutura usando o operador ponto por meio do nome da estrutura, seguido por um ponto e pelo nome do campo que se quer acessar, podemos acessar e visualizar o campo rua da estrutura x (endereco) usando a seguinte declaração: printf("%s", x.rua);

2. A estrutura de dados structs em linguagem C pode realizar diversas operações. Dentre as opções abaixo, identifique uma operação possível a ser realizada com uma struct.

A alternativa **"C "** está correta.

A estrutura de dados structs em linguagem C pode realizar diversas operações, como atribuição de variáveis da estrutura a variáveis da estrutura do mesmo tipo, leitura do endereço de uma variável de estrutura (operador &), acesso aos membros de uma variável de estrutura e uso do operador sizeof para determinar o tamanho de uma variável de estrutura. O acesso aos membros da estrutura pode ser feito através do operador ponto (.), se for feita a referenciação direta ou operador seta (->), se forem empregados ponteiros.

MÓDULO 4

⦿ **Empregar as estruturas de dados aninhadas, os vetores de estruturas e a instrução typedef usando a linguagem de programação C**

STRUCT ANINHADA

DEFINIÇÃO

Uma estrutura de dados aninhada ou struct aninhada é basicamente uma estrutura dentro de outra, ou seja, uma estrutura contida em outra ou uma estrutura que pode ser acessada por outra.

Desta maneira, podemos entender que a struct aninhada é um aninhamento de estruturas que nos permite criar estruturas que contenham outras estruturas internas.

O padrão ANSI C especifica que as estruturas de dados (struct) podem ser aninhadas até 15 níveis, mas a maioria dos compiladores permite mais.

Fonte: Shutterstock.

DECLARAÇÃO

Apenas para relembrar, sabemos que a declaração de estrutura declara um tipo struct. E cada tipo struct recebe um nome (ou tag) que se refere àquele tipo pelo nome precedido pela palavra struct. Cada unidade de dados na estrutura é chamada membro e possui um nome de membro. Os membros de uma estrutura podem ser de qualquer tipo. Sendo assim, os membros de estruturas podem ser também outras estruturas.

As estruturas aninhadas podem ser declaradas de duas formas.

A primeira forma é colocar uma estrutura dentro de outra, literalmente. Portanto, a sintaxe para trabalhar com estruturas aninhadas nessa forma é:

```
typedef struct {
	tipo membro_1;
	tipo membro_2;
	...
	tipo membro_n;
	struct{
		tipo membro_interno_1;
		tipo membro_interno _2;
		...
		tipo membro_interno _n;
	}
} Nome_estrutura;
Nome_estrutura NE;
```

E para acessar os membros, o acesso é direto e a sintaxe é:

```
//Para atribuir valores:
NE.membro_interno_1 = 0;
NE.membro_1 = 0;

//Para leitura do teclado:
&NE.membro_interno_1
&NE.membro_1

//Para impressão:
NE.membro_interno 1
NE.membro_1
```

A outra forma é você declarar a estrutura_1 antes e, na estrutura_2, declarar uma variável da estrutura 1. Desta forma, a sintaxe é a seguinte:

```
typedef struct {
	tipo membro_1;
	tipo membro_2;
	...
	tipo membro_n;
} nome_estrutura_1;
```

```
typedef struct {
        tipo membro_1;
        tipo membro_2;
        nome_estrutura_1 NE1;
        ...
        tipo membro_n;
} nome_estrutura_2;

nome_estrutura_2 NE2;
```

E para acessar os membros, a sintaxe é:

```
//para acessar os membros
NE2.membro1;
NE2.membro2;
NE2.NE1.membro1;
NE2.NE1.membro2;
```

Para entendermos melhor as duas formas de declaração de estrutura aninhadas, vamos analisar o Exemplo 7 abaixo.

Exemplo 7

```
1       #include < stdio.h >
2       #include < stdlib.h >
3
4       struct departamento {
5               int cod;
6               char descricao[30];
7       };
8
9       struct cargo {
10              int cod;
11              char descricao[30];
12      };
13
14      struct funcionario {
15              int cod;
16              char nome[30];
17              float salario;
18              struct departamento depto;
19              struct cargo cargo;
```

```
20  };
21
22  struct funcionario Funcionario;
23
24  int main(void)
25   {
26
27  }
```

Observe as linhas de 4 a 20. Nesse trecho, são definidas três estruturas que recebem os nomes: departamento, cargo e funcionário. A estrutura funcionário possui 5 membros, os dois últimos membros são do tipo struct departamento e struct cargo, isto é, dentro da estrutura funcionário, temos dois membros do tipo struct.

Em resumo, temos:

Nas linhas 4 e 7, foi declarada a estrutura departamento.

Nas linhas 9 e 12, foi declarada a estrutura cargo.

Nas linhas 14 e 20, foi declarada a estrutura funcionário.

Na linha 18, o campo depto é do tipo struct departamento.

Na linha 19, o campo cargo, da mesma forma que o campo depto, é do tipo struct cargo.

Ao analisar o código, visualizamos uma struct aninhada nas linhas 18 e 19, com os campos depto e cargo. Temos, então, um aninhamento de estruturas.

Outra forma de declarar uma estrutura é colocar uma estrutura dentro da outra, literalmente, como apresentado no Exemplo 8 a seguir:

Exemplo 8:

```
1    #include < stdio.h >
2   #include < stdlib.h >
3
4    struct cargo {
5          int cod;
6          char descricao[30];
7    };
8
9    struct funcionario {
10        int cod;
11         char nome[30];
```

```
12          float salario;
13           struct departamento {
14                  int cod;
15                   char descricao[30];
16          };
17
18           struct cargo cargo;
19   };
20
21    struct funcionario Funcionario;
22
23   int main(void)
24   {
25
26   }
```

Além das duas formas de declaração de estrutura aninhada mostradas anteriormente, pode-se utilizar também o tipo typedef ao invés de struct.

Sendo assim, vamos analisar o Exemplo 8 adaptado para o uso de typedef.

Exemplo 8 – adaptado:

```
1     #include < stdio.h >
2     #include < stdlib.h >
3
4      typedef struct departamento {
5            int cod;
6            char descricao[30];
7      } Departamento;
8
9     typedef struct cargo {
10           int cod;
11           char descricao[30];
12    } Cargo;
13
14    typedef struct funcionario {
15           int cod;
16           char nome[30];
17           float salario;
18            Departamento depto;
19            Cargo cargo;
20    } Funcionario;
21 printf( "\n TEXTO %_" , nome_vetor_struct[indice].nome_membro_struct);
```

```
22    int main(void)
23   {
24
25    }
```

Ao analisar o código, podemos observar que o typedef foi declarado em conjunto com o struct, tanto para a estrutura departamento quanto para cargo e para a estrutura funcionário. Isto é útil apenas para resumir a codificação, mas não influi no desempenho da aplicação.

💬 COMENTÁRIO

O comando typedef permite criar um novo nome para outro tipo de dados. Assim, o novo tipo de dados pode ser declarado como um tipo de dados primitivo existente no Linux.

Ao analisar a declaração dos campos depto e cargo, observe que estes campos agora não são mais do tipo estrutura, mas dos tipos definidos **Departamento** e **Cargo**, respectivamente.

Desta forma, pode-se declarar campos de outros tipos definidos dentro de uma estrutura que também pode ser um tipo definido. Além disso, não há uma limitação para o número de níveis na declaração aninhada, contudo, não seria prático incluir muitos níveis, pois isto irá complicar o entendimento para manutenção do código-fonte.

MANIPULAÇÃO DE STRUCTS ANINHADAS

Agora que já sabemos como declarar uma estrutura aninhada, vamos aprender como manipular os membros de uma struct aninhada.

Observe o trecho do programa que define duas estruturas, porém a segunda tendo membros que são também estruturas.

```
1      #define LEN 50
2
3      struct endereco {
4              char rua[LEN];
5             char cidade_estado_cep[LEN];
6      };
7
8      struct student {
9              char id[10];
10             int idade;
11                     struct endereco casa;
12             struct endereco escola;
```

```
13    };
14
15    struct student pessoa;
```

Dadas estas definições, é possível acessar os seguintes campos da variável pessoa do tipo struct (uma variável do tipo struct student).

pessoa.id
pessoa.casa.rua
pessoa.casa.cidade_estado_cep
pessoa.escola.rua
pessoa.escola.cidade_estado_cep

Note o uso repetido de "." quando se acessa membros dentro de membros.

MÃO NA MASSA

EXEMPLOS PRÁTICOS

No Exemplo 9, vamos criar uma struct para armazenar os dados do aluno (código, nome e data de nascimento).

```
1     #include < stdio.h >
2     #include < stdlib.h >
3     #include < string.h >
4     #include < locale.h >
5
6     typedef struct {
7        int codigo;
8        char nome[200];
9            struct {
10                int dia;
11                int mes;
12                int ano;
13           };
14    } Aluno;
15   Aluno aluno;
16
```

```
17          int main(void) {
18      setlocale(LC_ALL,"portuguese");
19      aluno.codigo = 0;
20      strcpy(aluno.nome, "NULL");
21      aluno.dia = 0;
22      aluno.mes = 0;
23      aluno.ano = 0;
24      printf(" \n O código do aluno é: %d ", aluno.codigo);
25      printf(" \n O nome do aluno é: %s ", aluno.nome);
26      printf(" \n A data de nascimento do aluno é: %d / %d / %d ", aluno.dia, aluno.mes, aluno.ano);
27      printf(" \n \n");
28      printf(" Digite o código do aluno: ");
29      scanf("%d%*c", &aluno.codigo);
30      printf(" Digite o nome do aluno: ");
31      scanf("%s%*c", &aluno.nome);
32      printf(" Digite o dia do nascimento do aluno: ");
33      scanf("%d%*c", &aluno.dia);
34      printf(" Digite o mês do nascimento do aluno: ");
35      scanf("%d%*c", &aluno.mes);
36      printf(" Digite o ano do nascimento do aluno: ");
37      scanf("%d%*c", &aluno.ano);
38      printf(" \n O código do aluno é: %d ", aluno.codigo);
39      printf(" \n O nome do aluno é: %s ", aluno.nome);
40      printf(" \n A data de nascimento do aluno é: %d / %d / %d ", aluno.dia, aluno.mes, aluno.ano);
41      printf(" \n \n");
42      system("pause");
43      return(0);
44  }
```

Observe as linhas de 06 a 14. Nesse trecho do código-fonte, é definida uma estrutura que recebe o nome de Aluno e, dentro dela, outra estrutura com os membros dia, mês e ano.

A estrutura que possui a data de nascimento do aluno está nas linhas 10, 11 e 12, note que ela não tem nome, e não é necessário que tenha, pois ela está contida dentro da estrutura Aluno. Podemos dizer que temos uma estrutura externa e uma estrutura interna, as quais são respectivamente Aluno e data.

Os membros da estrutura interna são acessados da mesma forma que a estrutura externa, isto é, diretamente.

Na linha 26, podemos observar a forma de acesso diretamente pelo nome da variável da Estrutura.

Agora, no Exemplo 10, vamos criar uma struct para armazenar os dados do aluno (código, nome e data de nascimento), porém, a declaração será diferente do Exemplo 9, pois alguns membros da estrutura são do tipo struct.

```
#include < stdio.h >
    #include < stdlib.h >
```

```
#include < string.h >
#include < locale.h >

typedef struct {
	int dia;
	int mes;
	int ano;
} Data;

typedef struct {
	int codigo;
	char nome[200];
	Data datNasc;
} Aluno;
Aluno aluno;

int main() {
	setlocale(LC_ALL,"portuguese");
	aluno.codigo = 0;
	strcpy(aluno.nome, "NULL");
	aluno.datNasc.dia = 0;
	aluno.datNasc.mes = 0;
	aluno.datNasc.ano = 0;
	printf(" \n O código do aluno é: %d ", aluno.codigo);
	printf(" \n O nome do aluno é: %s ", aluno.nome);
	printf(" \n A data de nascimento do aluno é: %d / %d / %d ", aluno.datNasc.dia, aluno.datNasc.mes, aluno.datNasc.ano);
	printf(" \n \n");
	printf(" Digite o código do aluno: ");
	scanf("%d%*c", &aluno.codigo);
	printf(" Digite o nome do aluno: ");
	scanf("%s%*c", &aluno.nome);
	printf(" Digite o dia do nascimento do aluno: ");
	scanf("%d%*c", &aluno.datNasc.dia);
	printf(" Digite o mês do nascimento do aluno: ");
	scanf("%d%*c", &aluno.datNasc.mes);
	printf(" Digite o ano do nascimento do aluno: ");
	scanf("%d%*c", &aluno.datNasc.ano);
	printf(" \n O código do aluno é: %d ", aluno.codigo);
	printf(" \n O nome do aluno é: %s ", aluno.nome);
	printf(" \n A data de nascimento do aluno é: %d / %d / %d ", aluno.datNasc.dia, aluno.datNasc.mes, aluno.datNasc.ano);
	printf(" \n \n");
	system("pause");
	return(0);
}
```

No Exemplo 10, as estruturas são declaradas separadamente. As estruturas que serão utilizadas dentro de outras devem ser declaradas antes.

📢 ATENÇÃO

Se você declarar uma estrutura e tentar utilizá-la dentro de outra estrutura, e a estrutura utilizada não tiver sido declarada antes, ao compilar o programa, um erro será apresentado.

Assim, em nosso exemplo, declaramos antes a estrutura data, que será usada dentro da estrutura aluno. Isto é notado nas linhas 6 a 16.

A forma de acesso aos membros da estrutura Aluno agora é um pouco diferente. Dentro da estrutura Aluno, é criada uma variável do tipo da estrutura Data, que é datNasc. Este nome, datNasc, é o nome que deverá ser utilizado para acessar os seus membros. Mas não somente ele, é necessário usar em conjunto com o nome da estrutura aluno, como mostram as linhas 23, 24 e 25.

ARRAY DE STRUCTS

DEFINIÇÃO

📷 Figura 15: Exemplo de vetor de estrutura (array de structs). Fonte: O Autor.

Ao analisar o vetor da figura 15, podemos verificar que cada posição do vetor armazena um conjunto de informações diferentes. Neste exemplo, cada posição está armazenando um tipo string no campo nome do aluno, um tipo de string no campo nome da disciplina e um tipo float no campo nota.

Sendo assim, podemos conceituar um vetor de estruturas ou um array de structs como uma estrutura de dados que armazena uma sequência de objetos, que são do mesmo tipo, e no caso, os objetos são estruturas.

Os arrays de structs são utilizados principalmente na criação de listas, podendo também ser usados em tarefas simples, como na repetição da leitura de dados de um cadastro.

DECLARAÇÃO

Para entendermos melhor a declaração de um array de structs, vamos voltar para o exemplo de vetor de estrutura ilustrado na Figura 16.

A declaração da estrutura do exemplo ficará da seguinte forma:

```
1 typedef struct {
2        char nome[200];
3        char disciplina[100];
4        float nota;
5 } Aluno;
6
7 Aluno aluno_nota[10];
```

Ao observarmos a declaração da estrutura, a linha 7 é a que cria de fato o vetor. No exemplo, é criado um vetor de 10 posições para armazenar o nome do aluno, a disciplina que o aluno está cursando e a nota do aluno.

INICIALIZANDO O ARRAY DE STRUCT

Para inicializar um array de struct, devemos atribuir aos seus membros os valores padrão (default) de cada tipo de dado. O código abaixo mostra como isto é feito no exemplo da Figura 21.

```
for(i=0; i< 10; i++) {
       strcpy(aluno_nota [i].nome, " NULL");
       strcpy(aluno_nota [i].disciplina = ‘ ‘);
       aluno_nota [i].nota = 0.0;
}
```

Observe que a função STRCPY da biblioteca string foi utilizada. Esta função copia um valor para uma variável de cadeia de caracteres.

POPULANDO O ARRAY DE STRUCT

Após a inicialização do array de struct, podemos inserir informações, dados. Sendo assim, é necessário um laço for para armazenar cada estrutura em uma posição. O código abaixo ilustra como armazenar as informações do nosso funcionário. Dentro do for, colocamos os comandos de leitura e escrita, necessários para pedir ao usuário o que ele deve digitar e, depois, armazenar o que foi digitado. Note que, para armazenar os dados que são inseridos a partir do teclado, precisamos utilizar a sintaxe:

```
&nome_vetor_struct[indice].nome_membro_struct;

For (i=0; i< 10; i++) {
   printf(" Digite nome do aluno: ");
   scanf("%s%*c", & aluno_nota [i].nome);
   printf(" Digite a disciplina do aluno: ");
   scanf("%s%*c", & aluno_nota [i].disciplina);
   printf(" Digite o nota do aluno: ");
   scanf("%f%*c", & aluno_nota [i].nota);
}
```

IMPRIMINDO O ARRAY DE STRUCT

No caso da impressão dos valores do array de struct, será necessário o uso do laço for para percorrer todas as posições desse array. A cada passagem no for, será impresso o conteúdo de uma posição do vetor na tela.

A sintaxe que você deve usar aqui é a seguinte:

printf("\n TEXTO %_" , nome_vetor_struct[indice].nome_membro_struct);

É necessária a inclusão do tipo de dado que deverá ser impresso, isto é, substitua %_ pelo correspondente (exemplo: %d para inteiro).

```
for(i=0; i< 10; i++) {
      printf(" \n O nome do aluno é: %d ", aluno_nota [i].nome);
      printf(" \n A disciplina do aluno é: %d ", aluno_nota [i].disciplina);
      printf(" \n A nota do aluno é: %.2f ", aluno_nota [i].nota);
}
```

BUSCANDO UM ELEMENTO NO ARRAY DE STRUCT

A pesquisa no array de struct pode procurar por um valor no vetor de estruturas. Por exemplo, você pode fazer uma pesquisa pelo nome, pela disciplina e pela nota, sendo assim, é possível fazer uma pesquisa por todos os membros da struct. Entretanto, é necessário tomar cuidado quando for buscar por uma string.

Vamos analisar o código de uma busca por uma string, neste caso, vamos procurar pelo membro nome.

```
printf(" \n Digite um nome: ");
```

```
scanf("%s%*c", nome);

for(i=0; i< 10; i++) {
    if(strcmp(nome, aluno_nota [i].nome)== 0) {
        printf("\n Registro encontrado! ");
        posicao = i;
    } else {
        posicao = -1;
    }
}

if(posicao = -1) {
    printf(" \n Registro não encontrado! ");
} else {
    printf(" \n Registro Encontrado: ");
    printf(" \n O nome do aluno é: %s ", aluno_nota [posicao].nome);
    printf(" \n A disciplina do aluno é: %s ", aluno_nota [posicao].disciplina);
    printf(" \n A nota do aluno é: %.2f ", aluno_nota [posicao].nota);
```

Agora, vamos analisar o código deste programa.

Primeiro, é solicitado ao usuário que digite um nome para a busca. Este nome será armazenado em uma variável chamada nome, que foi declarada anteriormente no programa.

Como o nome terá de ser procurado neste array, então um laço for será usado para percorrer esse array e analisar, posição a posição, se o nome digitado pelo usuário está ali ou não. Dessa forma, a string digitada pelo usuário será comparada com a string armazenada na posição i do vetor.

Uma função chamada strcmp(string1, string2); da biblioteca de strings é utilizada para fazer a comparação da primeira string com a segunda string. Aqui, a variável nome será comparada com o nome que está armazenado em cada uma das posições do vetor (aluno_nota [i].nome).

Quando o resultado dessa comparação for igual a zero, significa que a string foi encontrada em uma determinada posição do vetor.

Mas, se o resultado for diferente de zero, então, a string não foi encontrada. Portanto, um if é usado dentro do for para fazer essa verificação:

```
if(strcmp(nome, aluno_nota [i].nome)== 0)
```

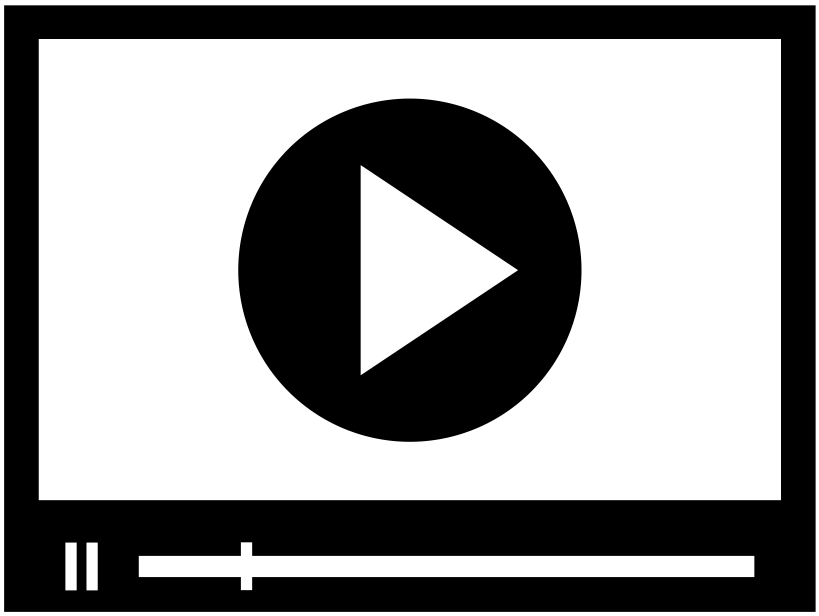
O if é adicionado logo depois do for para verificar se posição é -1 ou não. Se a posição for igual a -1, então o registro não foi encontrado, caso contrário, é impresso o valor da posição da estrutura. Para que isso seja possível, a variável i, que seria o índice do vetor, é substituída pela variável posição, que contém o valor exato da posição em que o nome foi encontrado.

MÃO NA MASSA

EXEMPLO PRÁTICO

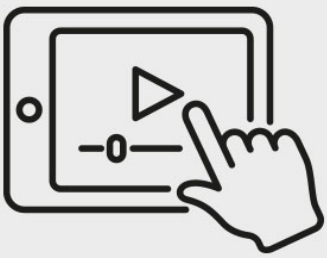
```
1 #include < stdio.h >
2 #include < stdlib.h >
3 #include < string.h >
4 #include < locale.h >
5
6 typedef struct {
7     int dia;
8     int mes;
9     int ano;
10 } Data;
11
12 typedef struct {
13     int codigo;
14     char nome[200];
15     Data datNasc;
16 } Aluno;
17
18 Aluno aluno[5];
19 int i=0;
20
21 int main() {
22     setlocale(LC_ALL,"portuguese");
23     for(i=0; i< 5; i++) {
24         aluno[i].codigo = 0;
25         strcpy(aluno[i].nome, "NULL");
26         aluno[i].datNasc.dia = 0;
27        aluno[i].datNasc.mes = 0;
28         aluno[i].datNasc.ano = 0;
29     }
30
31     for(i=0; i< 5; i++) {
32         printf(" \n ======================================================================");
33         printf(" \n O código do aluno é: %d ", aluno[i].codigo);
34         printf(" \n O nome do aluno é: %s ", aluno[i].nome);
35         printf(" \n A data de nascimento do aluno é: %d / %d / %d ",
36             aluno[i].datNasc.dia, aluno[i].datNasc.mes,
37             aluno[i].datNasc.ano);
38         printf(" \n ");
```

```
39    }
40
41    for(i=0; i< 5; i++) {
42        printf(" \n ========================================================================");
43        printf(" \n ========================================================================");
44        printf(" \n Digite o código do aluno: ");
45        scanf("%d%*c", &aluno[i].codigo);
46        printf(" \n Digite o nome do aluno: ");
47        scanf("%s%*c", &aluno[i].nome);
48        printf(" \n Digite o dia do nascimento do aluno: ");
49        scanf("%d%*c", &aluno[i].datNasc.dia);
50        printf(" \n Digite o mês do nascimento do aluno: ");
51       scanf("%d%*c", &aluno[i].datNasc.mes);
52       printf(" \n Digite o ano do nascimento do aluno: ");
53        scanf("%d%*c", &aluno[i].datNasc.ano);
54   }
55
56   for(i=0; i< 5; i++) {
57       printf(" \n ========================================================================");
58       printf(" \n O código do aluno é: %d ", aluno[i].codigo);
59       printf(" \n O nome do aluno é: %s ", aluno[i].nome);
60        printf(" \n A data de nascimento do aluno é: %d / %d / %d ",
61            aluno[i].datNasc.dia, aluno[i].datNasc.mes,
62           aluno[i].datNasc.ano);
63       printf(" \n ");
64   }
65   system("pause");
66   return(0);
67}
```



UTILIZANDO TYPEDEF E STRUCTS ANINHADAS

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



VERIFICANDO O APRENDIZADO

1. CONSIDERE QUE VOCÊ ESTÁ DESENVOLVENDO UM NOVO SOFTWARE PARA CATALOGAR OS JOGADORES DE FUTEBOL. VOCÊ MODELOU O SEU PROGRAMA UTILIZANDO AS STRUCTS APRESENTADAS ABAIXO.

```
TYPEDF STRUCT TIME {
   CHAR NOME[50];
} TIME;

STRUCT JOGADOR {
```

```
    INT NR_CAMISA;
    CHAR NOME[30];
    TIME T;
} JOGADOR;
```

CONSIDERANDO QUE VOCÊ PRECISE CADASTRAR UM NOVO JOGADOR CHAMADO MANOEL DA SILVA, QUE IRÁ ATUAR NO TIME DA ESTUDANTE ATLÉTICO CLUB, COM A CAMISA NÚMERO 10, QUAL ALTERNATIVA APRESENTA A FORMA CORRETA DE INSERIR OS DADOS?

A) Camisa->jogador = 10;

strcpy(nome->jogador, "Manoel da Silva");

strcpy(t->nome->jogador, " Estudante Atlético Club ");

B) jogador.camisa = 10;

strcpy(jogador.nome, "Manoel da Silva");

strcpy(jogador.t.nome, " Estudante Atlético Club ");

C) jogador.camisa = 10;

strcpy(jogador.nome, "Manoel da Silva");

strcpy(jogador.nome.nome, " Estudante Atlético Club ");

D) jogador->camisa = 10;

strcpy(jogador->nome, "Manoel da Silva");

strcpy(jogador->t->nome, " Estudante Atlético Club ");

E) camisa.jogador = 10;

strcpy(nome.jogador, "Manoel da Silva");

strcpy(t.nome.jogador, " Estudante Atlético Club ");

2. CONSIDERE O TRECHO DE CÓDIGO ABAIXO:

```
TYPEDEF STRUCT {
    CHAR NOME[200];
    INT IDADE;
    FLOAT SALARIO;
} FUNCIONARIO;
FUNCIONARIO FUNC[10];.
```

QUAL DAS OPÇÕES APRESENTA A SINTAXE CORRETA NA QUAL SEJA PREENCHIDO CORRETAMENTE O ELEMENTO DE UMA POSIÇÃO DO VETOR DE STRUCT?

A) &Funcionario[indice].idade;

B) &Funcionario[indice];

C) &func.idade[indice];

D) &func[indice].idade;

E) &func.idade;

GABARITO

1. Considere que você está desenvolvendo um novo software para catalogar os jogadores de futebol. Você modelou o seu programa utilizando as structs apresentadas abaixo.

```
typedf struct time {
   char nome[50];
} Time;

struct Jogador {
   int nr_camisa;
   char nome[30];
   Time t;
} jogador;
```

Considerando que você precise cadastrar um novo jogador chamado Manoel da Silva, que irá atuar no time da Estudante Atlético Club, com a camisa número 10, qual alternativa apresenta a forma correta de inserir os dados?

A alternativa **"B "** está correta.

A estrutura time está aninhada na estrutura jogador, sendo a estrutura time um tipo de dados definido pelo programador. Para acessar um elemento da estrutura jogador, basta acessar a variável referenciada diretamente e o nome do campo, utilizando o operador ponto. Para os campos camisa e nome, basta utilizar jogador.camisa e jogador.nome, sendo que, para o último caso, utilizando a função strcpy. Para acessar o nome do time, deve ser considerado o aninhamento da estrutura Time referenciada pela variável t. Portanto, deve-se utilizar jogador.t.nome, empregando a função strcpy.

2. Considere o trecho de código abaixo:

```
typedef struct {
   char nome[200];
   int idade;
   float salario;
} Funcionario;
Funcionario func[10];.
```

Qual das opções apresenta a sintaxe correta na qual seja preenchido corretamente o elemento de uma posição do vetor de struct?

A alternativa **"D "** está correta.

Para preencher corretamente o elemento de uma posição do vetor, é necessário indicar a posição do vetor de struct seguido pelo elemento a ser preenchido, portanto, a sintaxe correta é &nome_vetor_struct[indice].nome_membro_struct, isto é, para preencher o vetor, você deve selecionar o vetor, seu índice e o membro a que se refere na struct. Por exemplo: &func[i].idade

CONCLUSÃO

CONSIDERAÇÕES FINAIS

Acabamos de dar nossos primeiros passos através das estruturas de dados, que são recursos que permitem organizar as formas pelas quais os dados são tratados em um algoritmo.

Iniciamos nossa jornada apresentando como os dados são alocados em memória, utilizando funções específicas na linguagem, e referenciadas através dos ponteiros, recurso largamente utilizado em diversas linguagens de programação.

Prosseguindo na nossa jornada, descobrimos as estruturas de dados heterogêneas, chamadas de registro, que permitem que sejam armazenados conjuntos de dados de diferentes tipos, como se fosse uma única variável.

Descobrimos que, para utilizar os registros na linguagem C, são empregadas as structs (estruturas), que são compostas por diversos elementos que podem ser de tipos diferentes, inclusive contendo outras estruturas aninhadas.

Por fim, aprendemos como definir novos tipos de dados através do comando typedef e o emprego de vetores (arrays) de structs.

Para ouvir um *podcast* sobre o assunto, acesse a versão online deste conteúdo.

REFERÊNCIAS

SCHILDT, H. **C, completo e total. 3**. ed. São Paulo: Makron Books, 1996. cap. 5, p. 113.

EXPLORE+

Para saber mais sobre os assuntos tratados neste tema, pesquise na internet:

Canal Linguagem C Programação Descomplicada.

Clique aqui para baixar o código-fonte dos exemplos.

CONTEUDISTA

Daisy Cristine Albuquerque da Silva

CURRÍCULO LATTES